1. Команда проекта (роли)

**Проект** – это виды деятельности, выполняемые для создания продукта, а также ресурсы и документация.

Заинтересованные лица (stackeholders):

Заказчик

Спонсор

Пользователи

Менеджеры (management):

Проектный менеджер

Ресурсный менеджер

Delivery менеджер

Исполнители (project members):

Бизнес-аналитик

Разработчик

Инженер по качеству (тестировщик)

2. Низкое связывание и высокое зацепление

Низкое связывание:

«Степень связности» — мера неотрывности элемента от других

элементов (либо мера данных, имеющихся у него о них).

**Низкая связанность** является оценочной моделью, которая диктует, как распределить обязанности, которые необходимо поддерживать. Иными словами, это распределение ответственностей и данных, обеспечивающее взаимную независимость классов.

Класс с «низкой» связанностью:

* Имеет слабую зависимость от других классов;
* Не зависит от внешних изменений
* Прост для повторного использования

Высокое зацепление

Предметные области следует разделять по классам.

Зацепление класса — мера подобия предметных областей методов элемента:

• «Высокое» зацепление — сфокусированные подсистемы

(предметная область определена, управляема и понятна);

• «Низкое» зацепление — абстрактные подсистемы.

**Высокое зацепление класса** — это оценочная модель, направленная на удержание объектов должным образом сфокусированными, управляемыми и понятными. Высокое зацепление обычно используется для поддержания низкой связанности. Высокое зацепление означает, что обязанности данного элемента тесно связаны и сфокусированы.

И наоборот, **низкое зацепление** — это ситуация, при которой данный элемент имеет слишком много несвязанных обязанностей. Элементы с низким зацеплением часто страдают от того, что их трудно понять, трудно использовать, трудно поддерживать.

3. Шаблоны GRASP

GRASP (general responsibility assignment software patterns — общие

шаблоны распределения ответственностей) — шаблоны,

используемые в объектно-ориентированном проектировании для

решения общих задач по назначению ответственностей классам

и объектам.

• Низкое связывание (Low Coupling) : низкая связанность, отвечает за то, что бы объекты в системе знали друг о друге как можно меньш

• Высокое зацепление (High Cohesion)

• Информационный эксперт (Information Expert) *:* информационный эксперт, как следует из названия, занимается не чем иным, как предоставлением информации об объекте

• Создатель (Creator) : шаблон creator говорит нам какие условия должны соблюстись, что бы объекты верно порождали друг друга)

• Контроллер: этот шаблон отвечает за то, к кому именно должны обращаться вызовы из View)

• Чистая Фабрикация (Pure Fabrication): позволяет дизайну системы соответствовать принципам низкой связности и высокого зацепления.

• Посредник (Inderector): реализует низкую связность между классами, путем назначения обязанностей по их взаимодействию дополнительному объекту — посреднику.

• Полиморфизм (Polymorphysm): Шаблон позволяет обрабатывать альтернативные варианты поведения на основе типа. При этом, альтернативные реализации приводятся к обобщенному интерфейсу.

• Устойчивость к изменениям (Protected Variations): заключается в устранении точек неустойчивости, путем определения их в качестве интерфейсов и реализации для них различных вариантов поведения.

4. Виды шаблонов GoF

**Шаблоны проектирования GoF** – это многократно используемые решения широко распространенных проблем, возникающих при разработке ПО.

Выделяют следующие типы шаблонов: *порождающие* (шаблоны, облегчающие создание объектов способом, который наиболее соответствует ситуации), *структурные* (в общем случае, имеют дело с отношениями между объектами, облегчая их совместную работу) и *шаблоны поведения* (используются в коммуникации между объектами, делая её более лёгкой и гибкой)

Порождающие: Абстрактная Фабрика, Фабричный метод, Строитель, Одиночка, Прототип.

Некоторые структурные: Адаптер, Мост, Декоратор, Фасад, Легковес, Заместитель.

Некоторые шаблоны поведения: Цепочка Обязанностей, Команда, Хранитель, Состояние, Стратегия, Посетитель и т.д.

5. Шаблон Фабричный метод

**Фабричный метод** —порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Структура:

**Product** — продукт (определяет интерфейс объектов, создаваемых абстрактным методом)

**ConcreteProduct** — конкретный продукт (реализует интерфейс Product)

**Creator** — создатель (объявляет фабричный метод, который возвращает объект типа *Product)*

**ConcreteCreator** — конкретный создатель (переопределяет фабричный метод таким образом, чтобы он создавал и возвращал объект класса *ConcreteProduct)*

6. Основные диаграммы UML (перечислить и знать значение)

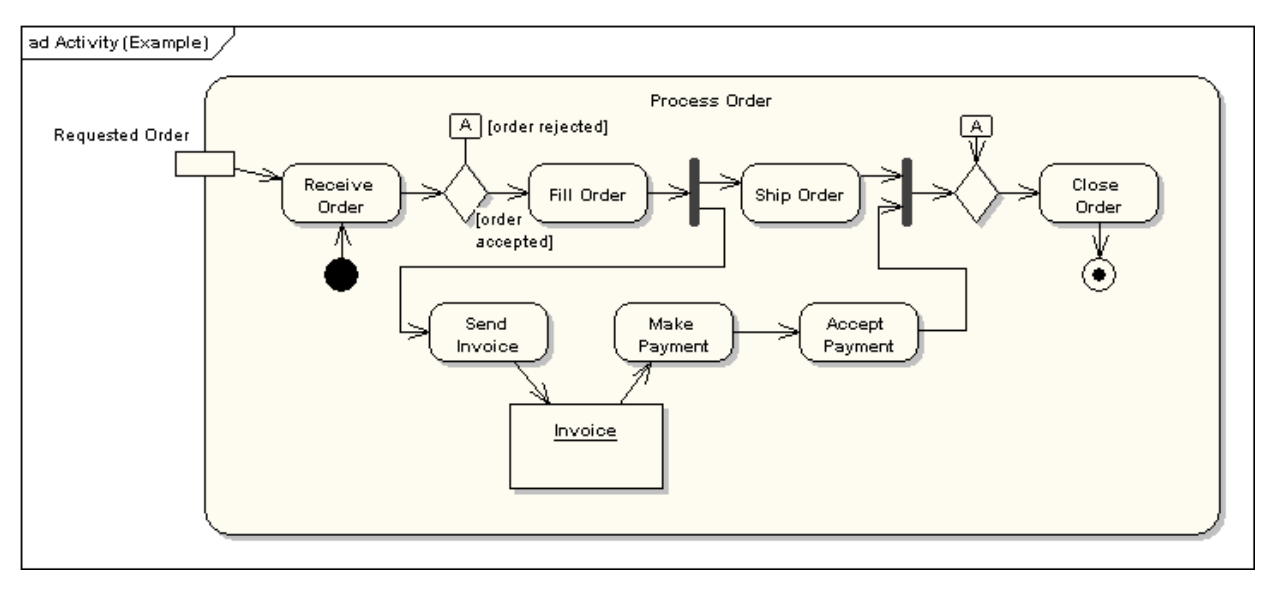
**Диаграмма последовательностей**

Диаграмма, отображающая последовательность, в которой объекты в процессе взаимодействия обмениваются сообщениями.

**Диаграмма действий**

диаграмма, на которой показано разложение некоторой деятельности на её составные части. Под деятельностью (англ. activity) понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий (англ. action), соединённых между собой потоками, которые идут от выходов одного узла к входам другого.

Диаграммы деятельности используются при моделировании бизнес-процессов, технологических процессов, последовательных и параллельных вычислений.



**Диаграмма состояний**

• Диаграмма перехода состояний позволяет получить точное, полное и ясное представление о механизме с конечным числом состояний.

• Диаграмма состояний – ориентированный граф для конечного автомата, в котором вершины обозначают состояние, а дуги показывают переход между двумя состояниями.

• Конечный автомат - это график состояний и переходов между ними, который описывает реакцию объекта.

**Диаграмма пакетов**

Диаграммы пакетов отображают зависимости между пакетами, составляющими модель.

**Диаграмма компонентов**

Диаграмма компонентов, Component diagram — статическая структурная диаграмма, показывает разбиение программной системы на структурные компоненты и связи (зависимости) между компонентами. В качестве физических компонентов могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т. п.

**Диаграмма развертывания**

Диаграмма развёртывания моделирует физическое развертывание артефактов на узлах.

**Диаграмма классов**

статическая структурная диаграмма, описывающая структуру системы, демонстрирующая классы системы, их атрибуты, методы и зависимости между классами.

Существуют разные точки зрения на построение диаграмм классов в зависимости от целей их применения:

концептуальная точка зрения — диаграмма классов описывает модель предметной области, в ней присутствуют только классы прикладных объектов;

точка зрения спецификации — диаграмма классов применяется при проектировании информационных систем;

точка зрения реализации — диаграмма классов содержит классы, используемые непосредственно в программном коде (при использовании объектно-ориентированных языков программирования).

7. Стандарты ТЗ

*ГОСТ 34.602-89*

Если заказчик требует оформления документации в соответствии с государственным стандартом, выбор делается в сторону стандарта ГОСТ 34.602-89. Подготовка Технического задания по ГОСТ 34.602-89 требует значительных временных затрат.

*IEEE Std 830*

Если поставлены сжатые сроки подготовки ТЗ и заказчик не требует оформления документации в соответствии с государственным стандартом, то можно использовать шаблон технического задания по стандарту IEEE Std 830. Стандарт IEEE Std 830 рекомендуют различные способы структурирования детальных требований для различных классов систем, поскольку не существует оптимальной структуры для всех систем. *Корпоративный шаблон*

Существует и третья альтернатива для выбора шаблона Технического задания, когда заказчик предлагает использовать принятый в компании Корпоративный шаблон для описания требований к информационным системам, избрав приоритетные для себя пункты

8. Структура ТЗ

Техническое задание – технический документ (спецификация), оговаривающий набор требований к системе и утверждённый как заказчиком/пользователем, так и исполнителем/производителем системы. Такая спецификация может содержать также системные требования и требования к тестированию.

Структура:

1. Введение.

Цель (или назначение системы).

Краткая сводка возможностей (в виде списка).

2. Обзор вариантов использования

Диаграмма вариантов использования

Описание актеров

3. Детальные требования

3.1 Описание вариантов использования

3.2 Пользовательский интерфейс

Эскизы основных форм

3.3 Нефункциональные требования

Ограничения (язык, операционная система, ограничение по памяти и пр.)

Надёжность (отказоустойчивость)

Безопасность

3.4 Архитектура системы

Диаграмма классов

Диаграмма базы данных

Диаграмма последовательностей (на один из вариантов использования)

Диаграмма состояний (например, состояние заказа, не надо описывать действия

пользователя)

Диаграмма действий (на один из вариантов использования)

Диаграмма компонентов (модули, можно выделить слои, звенья)

4. Порядок проведения приемочных испытаний

Тестовые сценарии (3-4 желательно чтобы они покрывали несколько вариантов использования)

5. Календарный план и стоимость проекта

9. Слои и уровни приложения

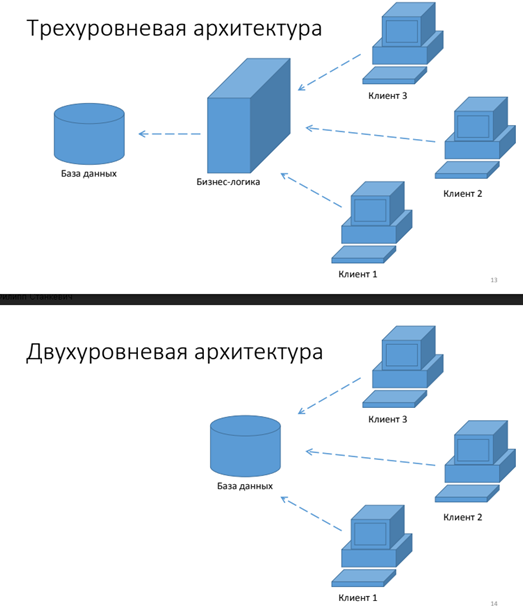
Слой (layer) – описывает логическую группировку модулей и

компонент в приложении



Уровень (tier) – описывает физическое распределение модулей и

компонент по серверам и компьютерам



**10. Шаблон DAO**

**DAO** - это объект, который предоставляет абстрактный интерфейс к какому-либо типу БД или механизму хранения. Этот шаблон проектирования применим ко множеству языков программирования, большинству программного обеспечения, нуждающемуся в хранении информации и к большей части баз данных.

**DAO** абстрагирует сущности системы и делает их отображение на БД, определяет общие методы использования соединения, его получение, закрытие и (или) возвращение в **Connection Pool**.

Вершиной иерархии **DAO** является абстрактный класс или интерфейс с описанием общих методов, которые будут использоваться при взаимодействии с базой данных. Как правило, это методы поиска, удаление по ключу, обновление и т.д

**11. Что такое Сервисы?**

Сервис – это видимый ресурс, выполняющий повторяющуюся задачу

и описанный внешней инструкцией

Ключевые идеи, которые стоят за понятием сервиса:

**Ориентация на бизнес**: поддерживается анализом сервиса и техникой проектирования

**Инструкции**: сервисы самодостаточны и описываются в терминах интерфейсов, операций, семантики, динамических характеристик, политик и свойств сервиса

**Повторное использование**: повторное использование сервисов обеспечивается их модульным планированием

**Соглашения**: сервисные соглашения заключаются между сущностями, именуемыми поставщиками и пользователями

**Размещение и видимость**: в течение своего жизненного цикла сервисы размещаются и становятся видимы благодаря сервисным метаданным, реестрам и хранилищам

**Агрегация**: на слабо связанных сервисах строятся объединяющие бизнес-процессы и сложные приложения для одного или нескольких предприятий

12. Шаблоны SOLID (перечислить и знать значение)

Принципы SOLID

Эти принципы, когда применяются вместе, предназначены для повышения вероятности того, что программист создаст систему, которую будет легко поддерживать и расширять в течение долгого времени.

Принципы SOLID — это руководства, которые могут применяться во время работы над программным обеспечением для удаления «кода с запашком» предписывая программисту выполнять рефакторинг исходного кода, пока тот не станет разборчиво написанным и расширяемым.

S – Принцип единственной ответственности

O – Принцип открытости/закрытости

L – Принцип подстановки Барбары Лисков

I – Принцип разделения интерфейса

D – Принцип инверсии зависимостей

S: Существует лишь одна причина, приводящая к изменению класса.

Каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс.

O: Программные сущности должны быть (классы, модули, функции и т. п.) открыты для расширения, но закрыты для модификации.

• открыты для расширения: поведение сущности может быть расширено, путём создания новых типов сущностей.

• закрыты для изменения: в результате расширения поведения сущности, не должны вноситься изменения в код, который эти сущности используют.

Идея была в том, что однажды разработанная реализация класса в дальнейшем требует только исправления ошибок, а новые или изменённые функции требуют создания нового класса.

L:Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.

Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом.

Этот принцип предупреждает разработчика о том, что изменение унаследованного производным типом поведения очень рискованно.

I: клиенты не должны зависеть от методов, которые они не используют

или лучше много маленьких интерфейсов, чем один большой.

D:

• Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракции.

• Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

13. Виды тестирования (перечислить и знать значения)

Функциональное тестирование — это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности ПО в определённых условиях решать задачи, нужные пользователям. Функциональные требования определяют, что именно делает ПО, какие задачи оно решает.

Дымовое тестирование в тестировании программного обеспечения означает минимальный набор тестов на явные ошибки. «Дымовой тест» обычно выполняется самим программистом; не проходившую этот тест программу не имеет смысла отдавать на более глубокое тестирование.

Приемочное тестирование – это комплексное тестирование, необходимое для определения уровня готовности системы к последующей эксплуатации. Тестирование проводится на основании набора тестовых сценариев, покрывающих основные бизнес-операции системы.

Тестирование пользовательского интерфейса - это процесс тестирования графического пользовательского интерфейса продукта, чтобы обеспечить его соответствие спецификациям.

Обычно это осуществляется с помощью различных тестовых сценариев.

Ручное тестирование — часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно производится тестировщиком без использования программных средств, для проверки программы или сайта путём моделирования действий пользователя. В роли тестировщиков могут выступать и обычные пользователи, сообщая разработчикам о найденных ошибках.

Автоматизированное тестирование программного обеспечения — часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно использует программные средства для выполнения тестов и проверки результатов выполнения, что помогает сократить время тестирования и упростить его процесс.

Модульное тестирование, или юнит-тестирование — процесс в программировании, позволяющий проверить на корректность отдельные модули (классы) исходного кода программы.

Интеграционное тестирование — одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию.

14. Шаблон Фасад

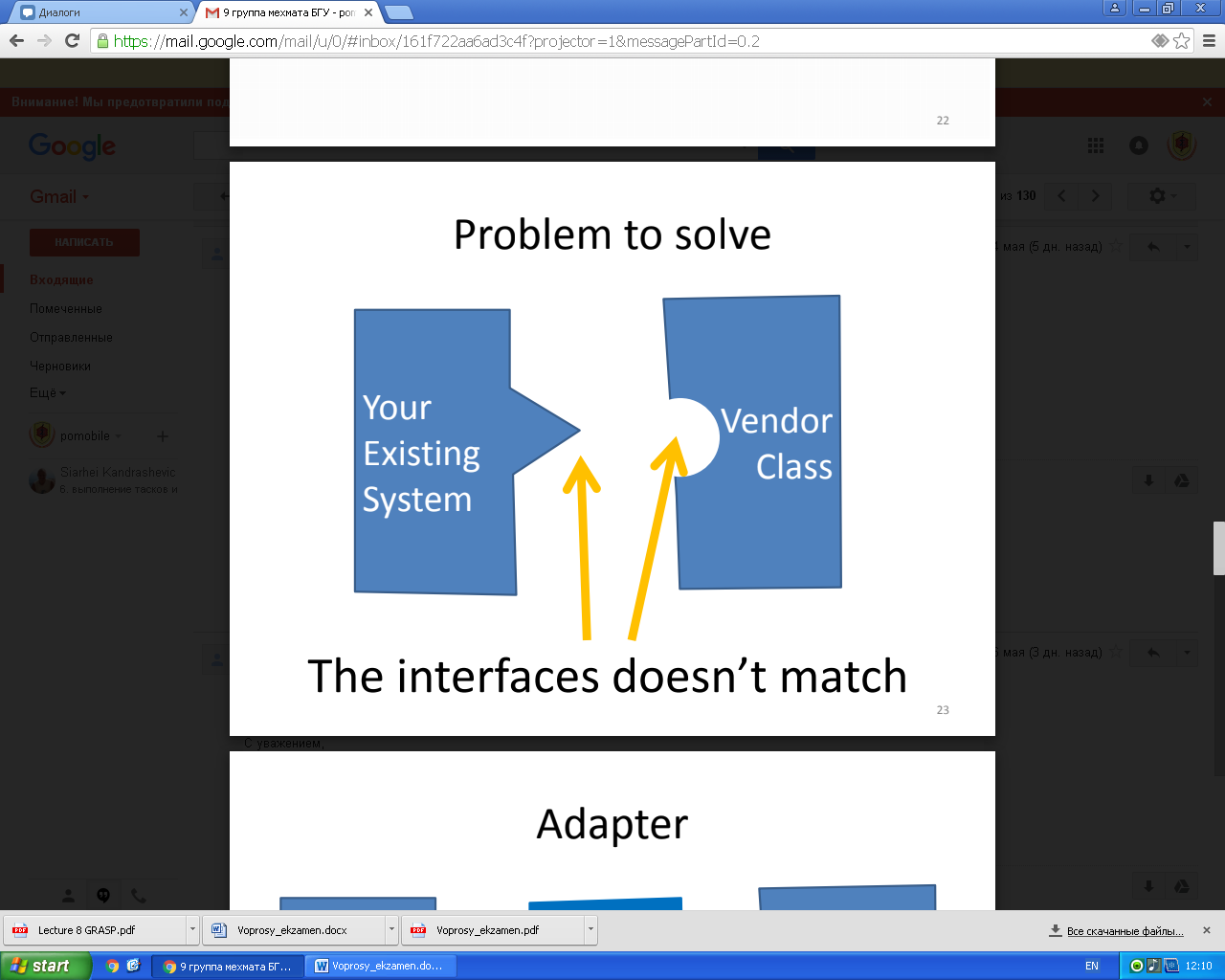
Фасад представляет собой шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней. не запрещает прямой доступ к подсистеме, просто он делает его проще и понятнее

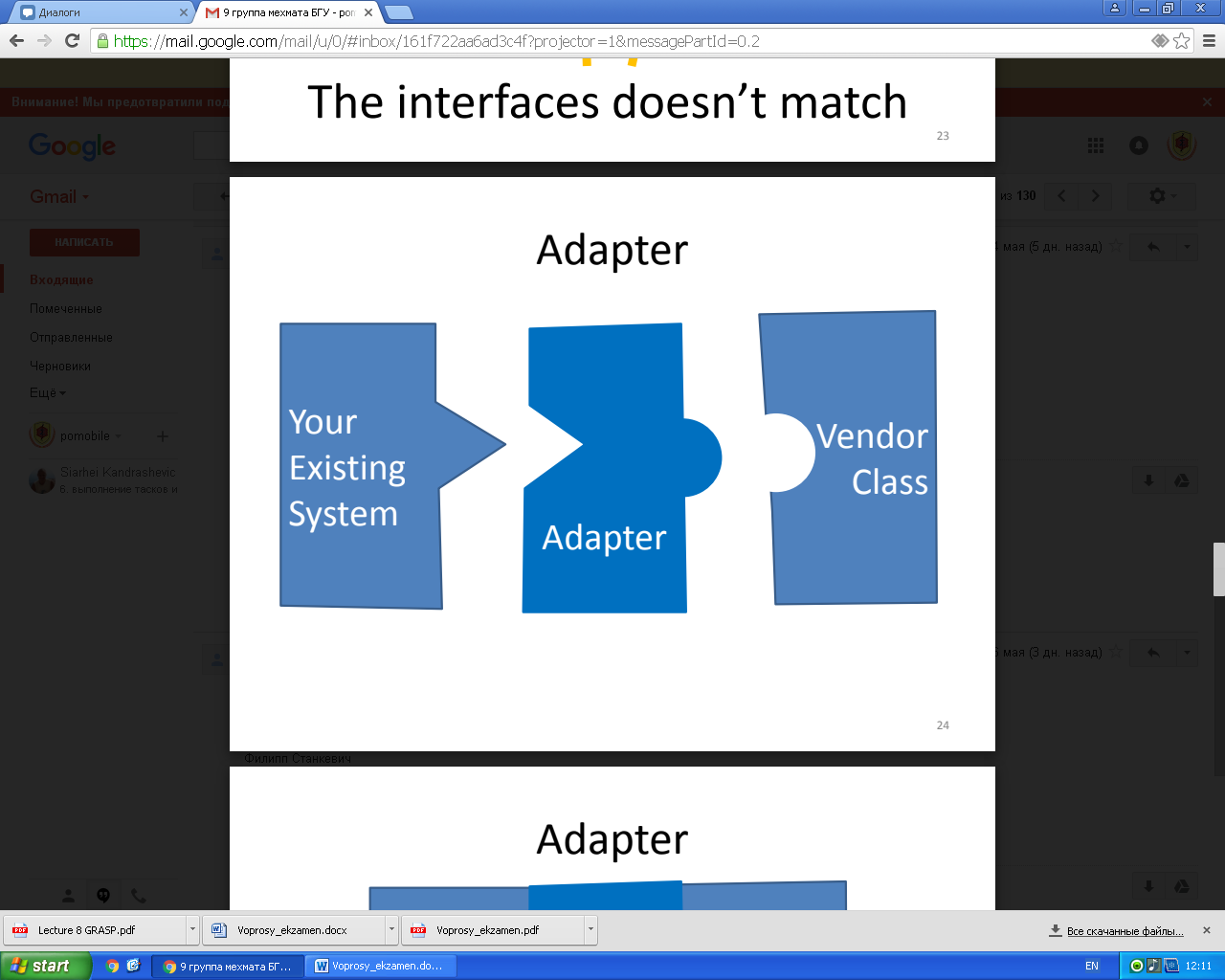
Решаемая проблема: Клиенты хотят получить упрощенный интерфейс к общей функциональности сложной подсистемы.

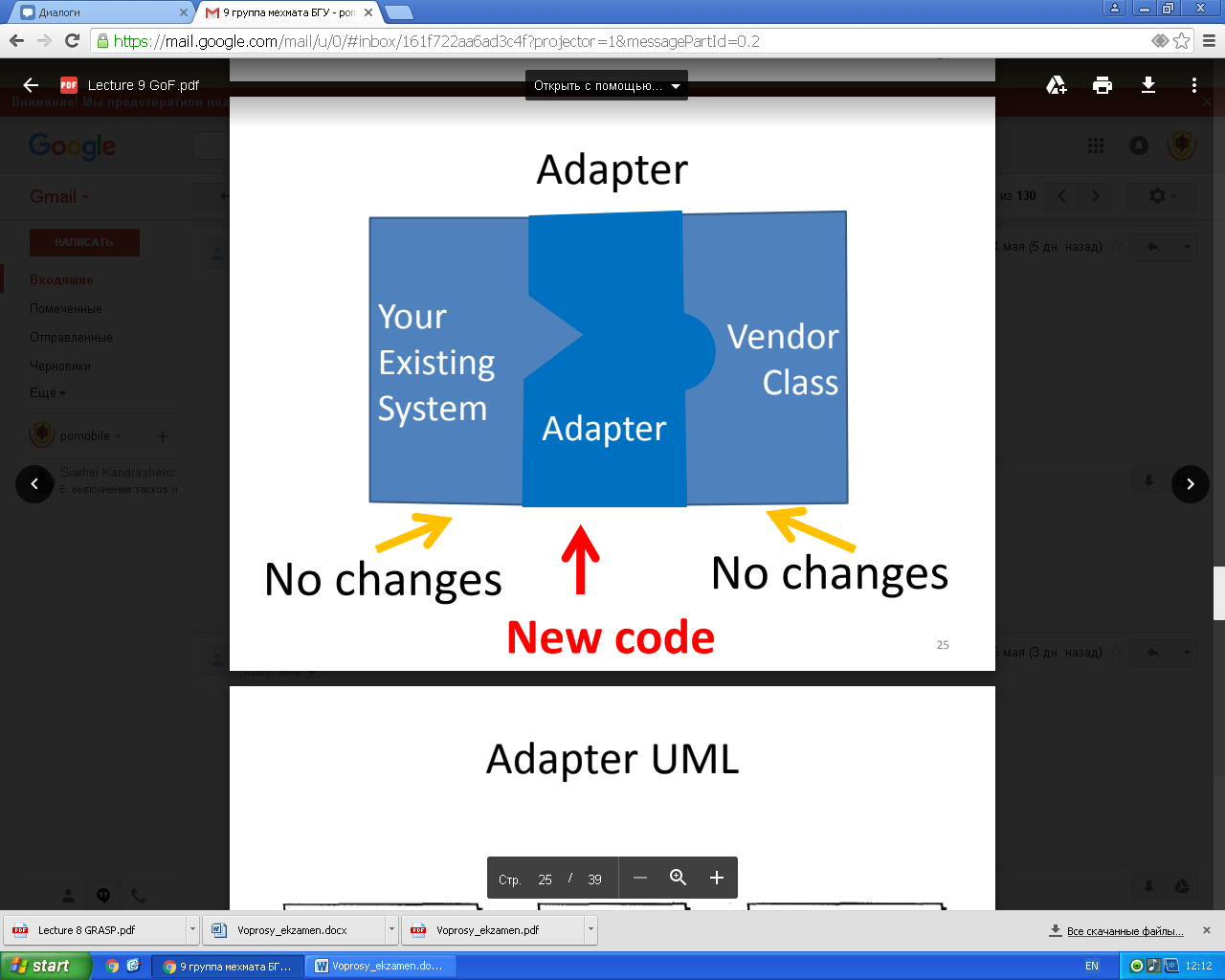
Пример фасада:

Потребитель звонит на один номер и разговаривает с представителем службы поддержки клиентов. Представитель службы поддержки клиентов выступает в качестве фасада, предоставляя интерфейс отделу исполнения заказов, отделу биллинга и отделу доставки.

15. Шаблон Адаптер







**Адаптер** – структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Задача: Система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс

Способ решения: Адаптер предусматривает создание класса-оболочки с требуемым интерфейсом.

Следствие: Шаблон Адаптер позволяет включать уже существующие объекты в новые объектные структуры, независимо от различий в их интерфейсах

16. Rational Unified Process

RUP – методология разработки программного обепечения, созданная компанией Rational Software. RUP обеспечивает строгий подход к распределению задач и ответственности внутри организации-разработчика. Его предназначение заключается в том, чтобы гарантировать создание точно в срок и в рамках установленного бюджета качественного ПО, отвечающего нуждам конечных пользователей.

Состоит из 4 фаз:

*Начальная стадия*

* Формируются видение и границы проекта
* Создается экономическое обоснование
* Определяются основные требования, ограничения и ключевая функциональность продукта
* Создается базовая версия модели прецедентов
* Оцениваются риски

При завершении начальной фазы оценивается достижение *этапа* *жизненного цикла* *цели*, которое предполагает соглашение заинтересованных сторон о продолжении проекта.

*Уточнение*

В фазе «Уточнение» производится анализ предметной области и построение исполняемой архитектуры

*Построение*

В фазе «Построение» происходит реализация большей части функциональности продукта.

*Внедрение*

В фазе «Внедрение» создается финальная версия продукта и передается от разработчика к заказчику. Это включает в себя программу бета-тестирования, обучение пользователей, а также определение качества продукта. В случае, если качество не соответствует ожиданиям пользователей или критериям, установленным в фазе Начало, фаза Внедрение повторяется снова. Выполнение всех целей означает достижение вехи готового продукта (Product Release) и завершение полного цикла разработки.

17. Гибкие методологии разработки ПО (общий смысл, перечислить)

**Гибкая методология разработки** — серия подходов к разработке ПО, ориентированных на использование итеративной (выполнение работ параллельно с непрерывным анализом полученных результатов и корректировкой предыдущих этапов работы) разработки, динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля.

Принципы гибкой разработки

Гибкие подходы основываются на следующих базовых принципах :

1. Работающее ПО является основным средством измерения развития проекта.

2. В таком случае, для развития проекта требуется, чтобы ПО разрабатывалось и

предоставлялось быстро путем добавления небольших улучшений.

3. Даже поздние изменения в требованиях должны быть приняты (модель

разработки на основе небольших приращений функциональности помогает их

принимать).

4. Личное общение является более предпочтительным, чем обмен документами.

5. Непрерывная обратная связь с потребителем и вовлечение потребителя в разработку является необходимым для создания высококачественного ПО.

6. Простое проектирование, которое развивается и улучшается со временем, является лучшим подходом в сравнении с выполнением заранее детального проектирования, которое учитывает все возможные сценарии использования создаваемого ПО.

7. Сроки передачи готового ПО определяются квалифицированными командами талантливых разработчиков (а не путем распоряжений).

Разновидности гибких подходов

• XP,

• Scrum,

• Kanban,

• Agile,

• Lean,

• Theory of Constraints,

• System Thinking,

• Flow-Based Product Development

Etc.

18. Итеративные методологии разработки ПО

Основная идея – пошаговая разработка ПО. На каждом шаге к создаваемой ПС добавляются некоторые функциональные возможности (до тех пор, пока система не будет реализована полностью). На первом шаге данной модели создается простое начальное приложение, реализующее подмножество задач полного решения проблемы.

• Оно должно включать некоторые из основных задач рассматриваемой проблемы

• которые легко понять и реализовать,

• которые формируют полезную и простую в применении систему.

Достоинства итеративной модели разработки

• Уменьшаются затраты на приспособление к изменившимся требованиям к ПО.

• Уменьшается объем анализа и кол-во документации, которые должны быть переделаны.

• Легче получить отзыв потребителя на выполненные результаты разработки.

• Потребитель может комментировать показываемое ему ПО и видеть, насколько много было разработано.

• Потребителю трудно оценивать ход разработки на основе документов проектирования ПО.

• Возможно ускорить предоставление и развертывание полезного ПО у потребителя, даже если не вся требуемая функциональность в нем имеется.

• Отсутствует риска получить «все или ничего».

Недостатки итеративной модели разработки

• Очень долгое время отсутствует целостное понимание возможностей и ограничений проекта.

• При итерациях приходится отбрасывать часть сделанной ранее работы.

• Снижается добросовестность специалистов при выполнении работ

• это психологически объяснимо, ведь над ними постоянно довлеет ощущение, что «всё равно всё можно будет переделать и улучшить позже».

19. Жизненный цикл программного продукта

**Жизненный цикл программного продукта**– это период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации

Стадии:

1. Выработка системных требований
2. Выработка требований к ПО
3. Анализ
4. Проектирование
5. Кодирование
6. Тестирование
7. Эксплуатация

20. Водопадная модель разработки ПО (стадии)

Модель водопада

1. **Выявление требований** - Отчет о реализуемости

2. **Анализ требований и планирование проекта** - Спецификация требований и план проекта

3. **Проектирование** - Описание проекта системы

4. **Кодирование** - Программный код

5. **Тестирование** - План и отчет о тестировании, руководства

6. **Установка** - Отчет об установке

7. **Сопровождение и развитие**

Преимущества модели водопада

• простота реализации (основное преимущество);

• получение полной и согласованной документации на каждом этапе;

• легкость определения сроков и затрат на проект;

• простота администрирования ходом выполнения проекта

• каждый этап завершается и создается его результат,

• некоторое количество денег передается заказчиком разрабатывающей организации.

Недостатки модели водопада

1. Предполагается, что требования к разрабатываемой системе остаются неизменными (замораживаются) до окончания разработки.

2. Замораживание требований обычно требует выбора технического обеспечения

• это составляет часть спецификации требований.

3. Вызывает подход «большого взрыва»

• полное ПО предоставляется за один раз, в конце разработки.

4. Поощряется «раздувание требований».

5. Данная модель является управляемой документами

• требует создание формального документа в конце каждого этапа.

• Несмотря на эти недостатки, модель водопада была (и часто остается) одной из наиболее широко используемых моделей процесса разработки ПО.

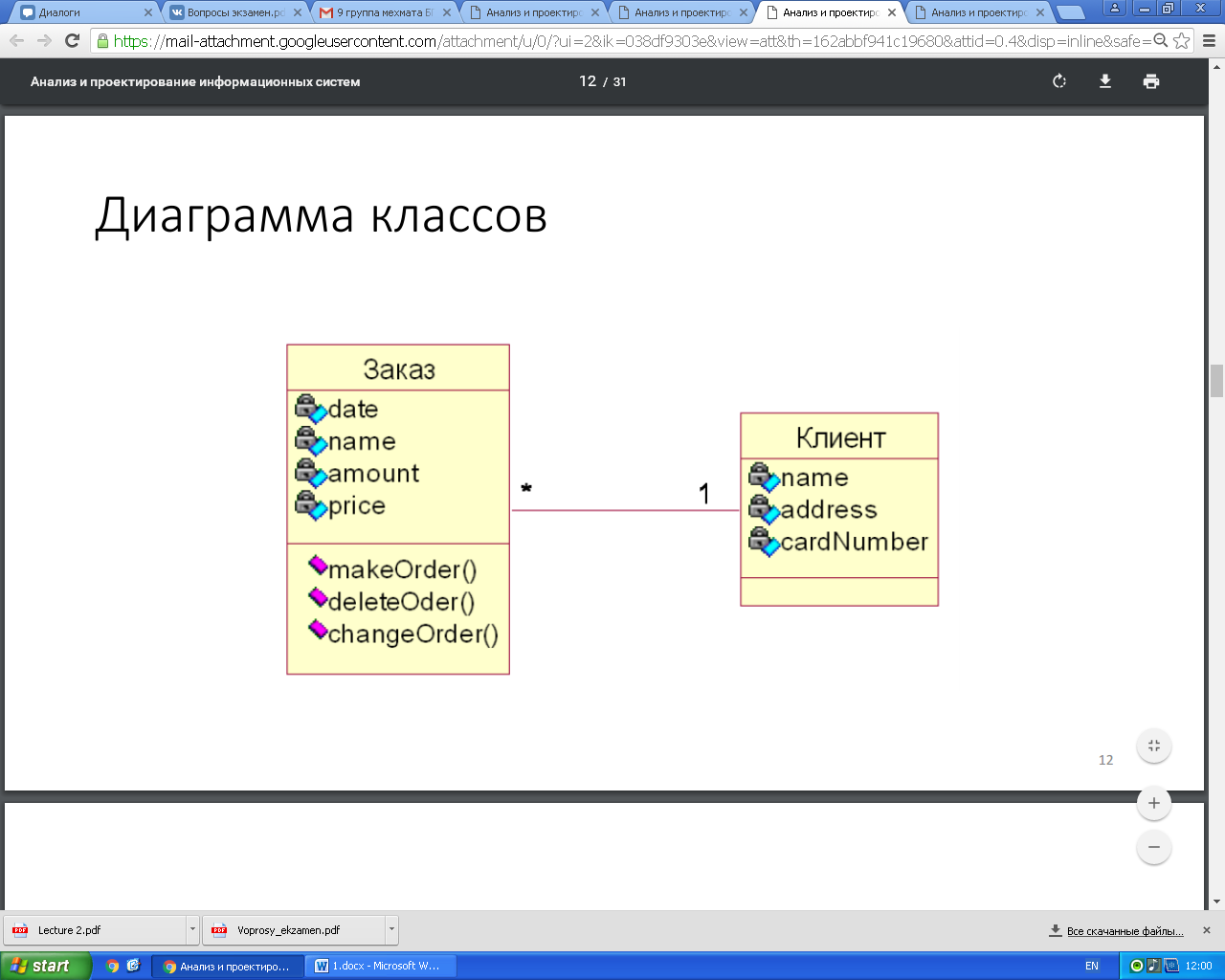
• Она хорошо подходит для рутинных проектов, в которых хорошо понятны требования.

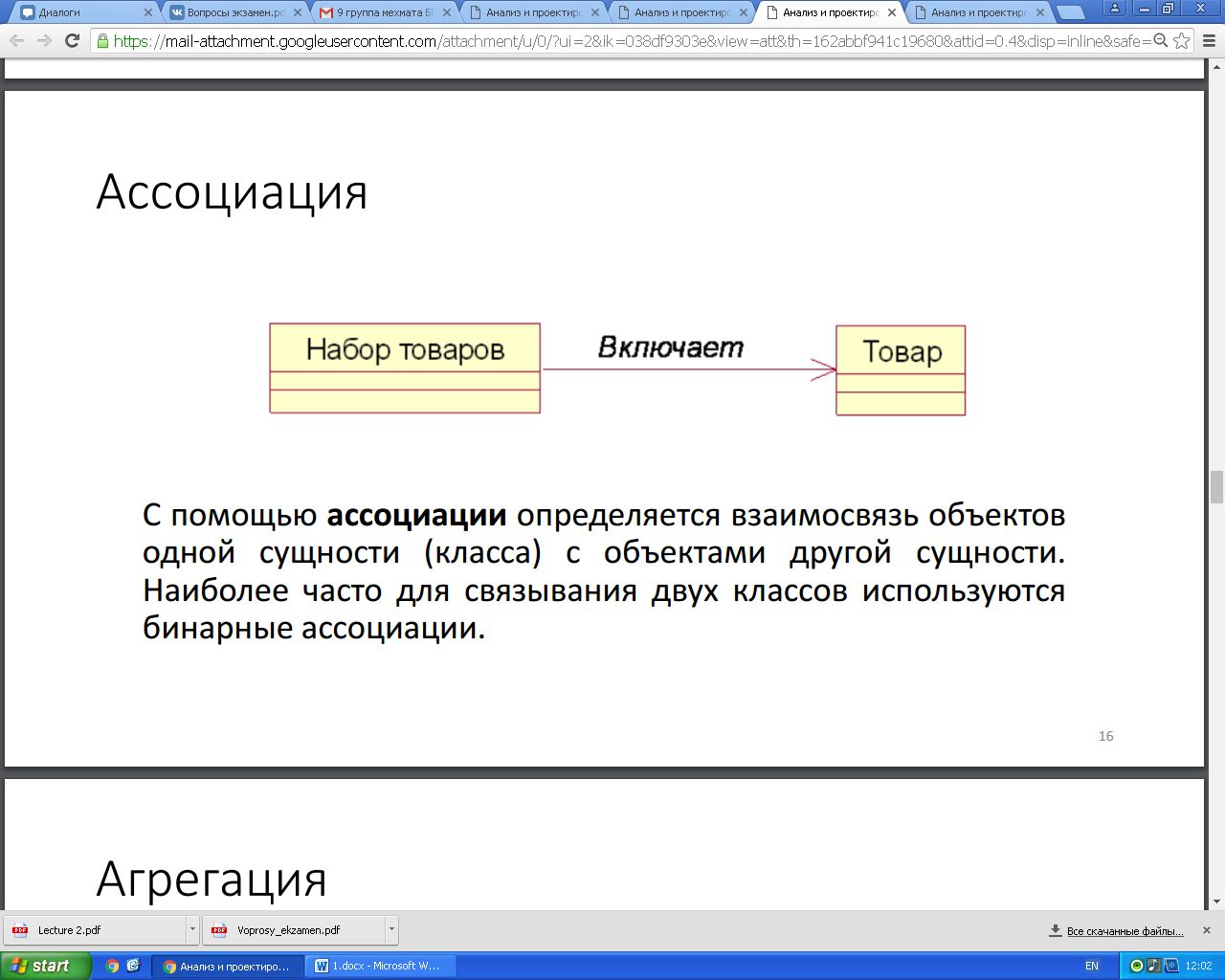
• Модель водопада хорошо работает и может быть наиболее эффективным процессом, если:

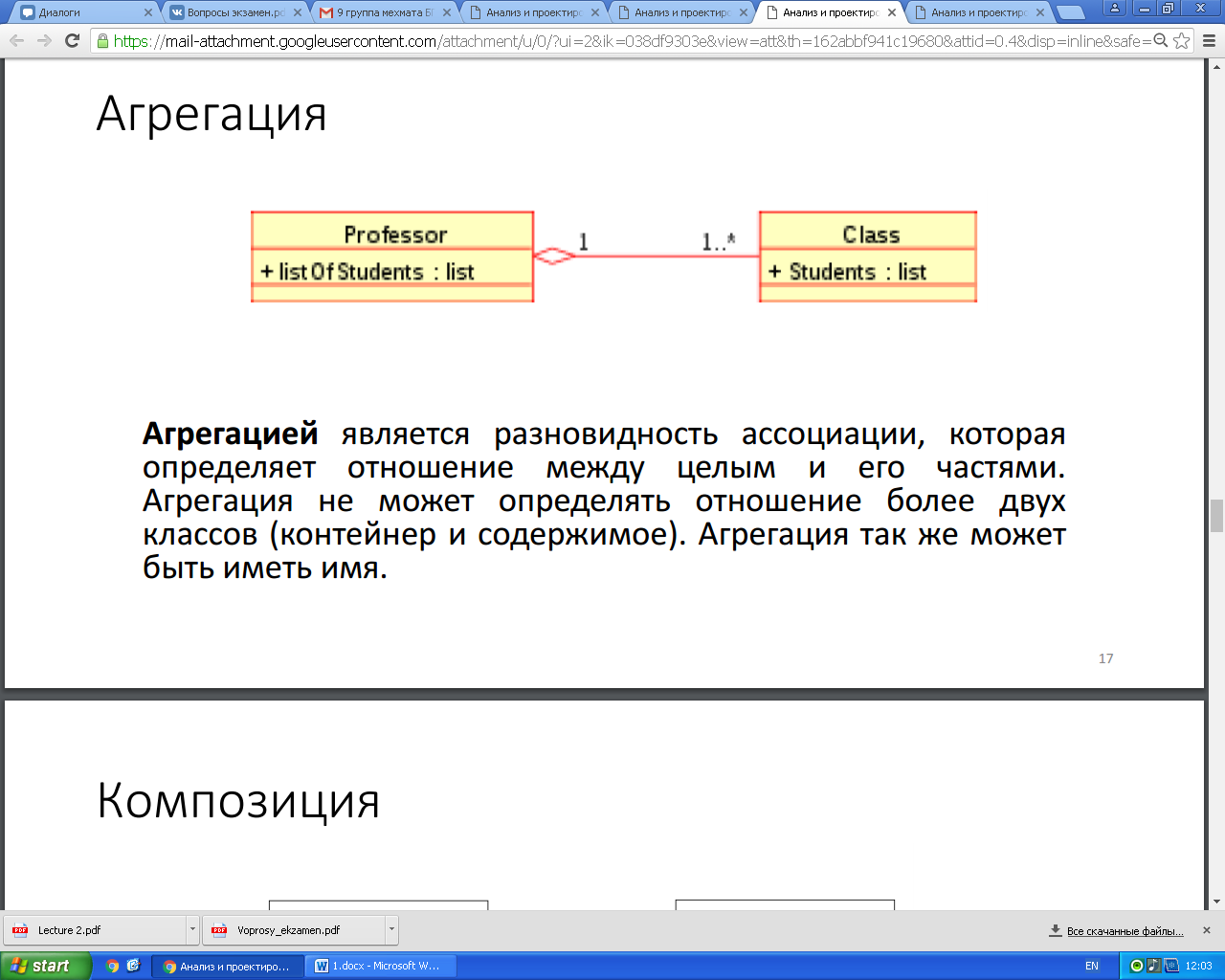
1. разрабатывающая организация достаточно хорошо знакома с предметной областью,

2. требования к ПО являются достаточно ясными

21. Диаграмма классов UML





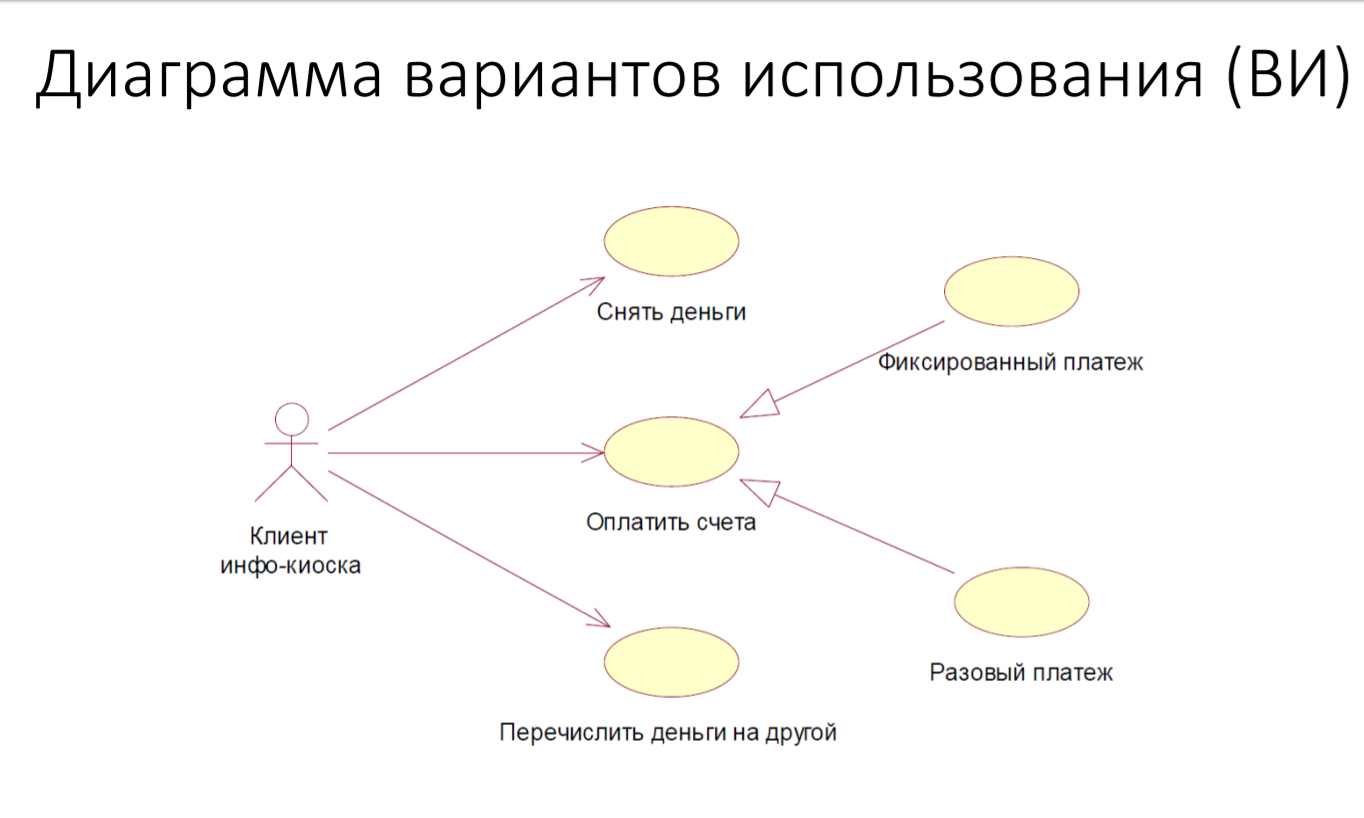




22. Диаграмма вариантов использования

Диаграмма вариантов использования (сценариев поведения, прецедентов) является исходным концептуальным представлением системы в процессе ее проектирования и разработки. Данная диаграмма состоит из актеров, вариантов использования и отношений между ними. При построении диаграммы могут использоваться также общие элементы нотации: примечания и механизмы расширения.

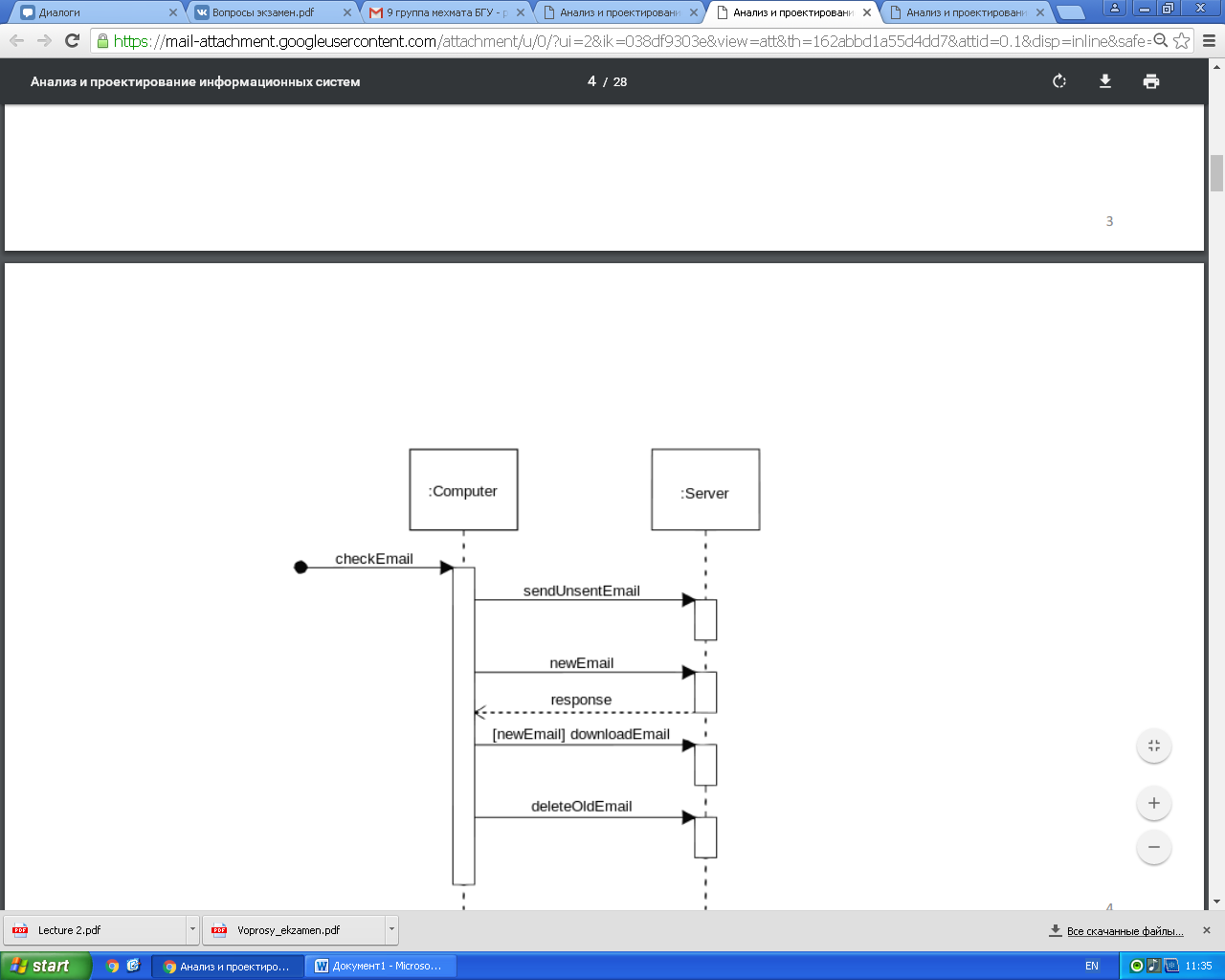
Суть данной диаграммы состоит в следующем [[28](https://sites.google.com/site/anisimovkhv/learning/pris/lecture/literatura#lit28)]: проектируемая система представляется в виде множества актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом ***актером***(действующим лицом, актантом, актором) называется любой объект, субъект или система, взаимодействующая с моделируемой системой извне. В свою очередь ***вариант использования*** – это спецификация сервисов (функций), которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемых системой при взаимодействии с актером. При этом в модели никак не отражается то, каким образом будет реализован этот набор действий.





23. Диаграмма последовательностей

Диаграмма, отображающая последовательность, в которой объекты в процессе взаимодействия обмениваются сообщениями.



*Синхронное сообщение* - актёр-отправитель передаёт ход управления актёру-

получателю, которому необходимо провести в прецеденте некоторое действие.

Пока проводимое актёром-получателем действие не будет завершено

(соответственно, не будет получено ответное сообщение), актёр-отправитель теряет

возможность производить какие-либо действия. Графически изображается как

стрелка с закрашенным треугольником, после которой идёт прямоугольник,

отражающий деятельность объекта, в конце которого находится ответное

сообщение.

*Ответное сообщение* - данное сообщение является ответом на синхронное

сообщение. Обычно, содержит какое-либо возвращаемое изначальному актёру-

отправителю значение, также возвращающее ему управление (возможность

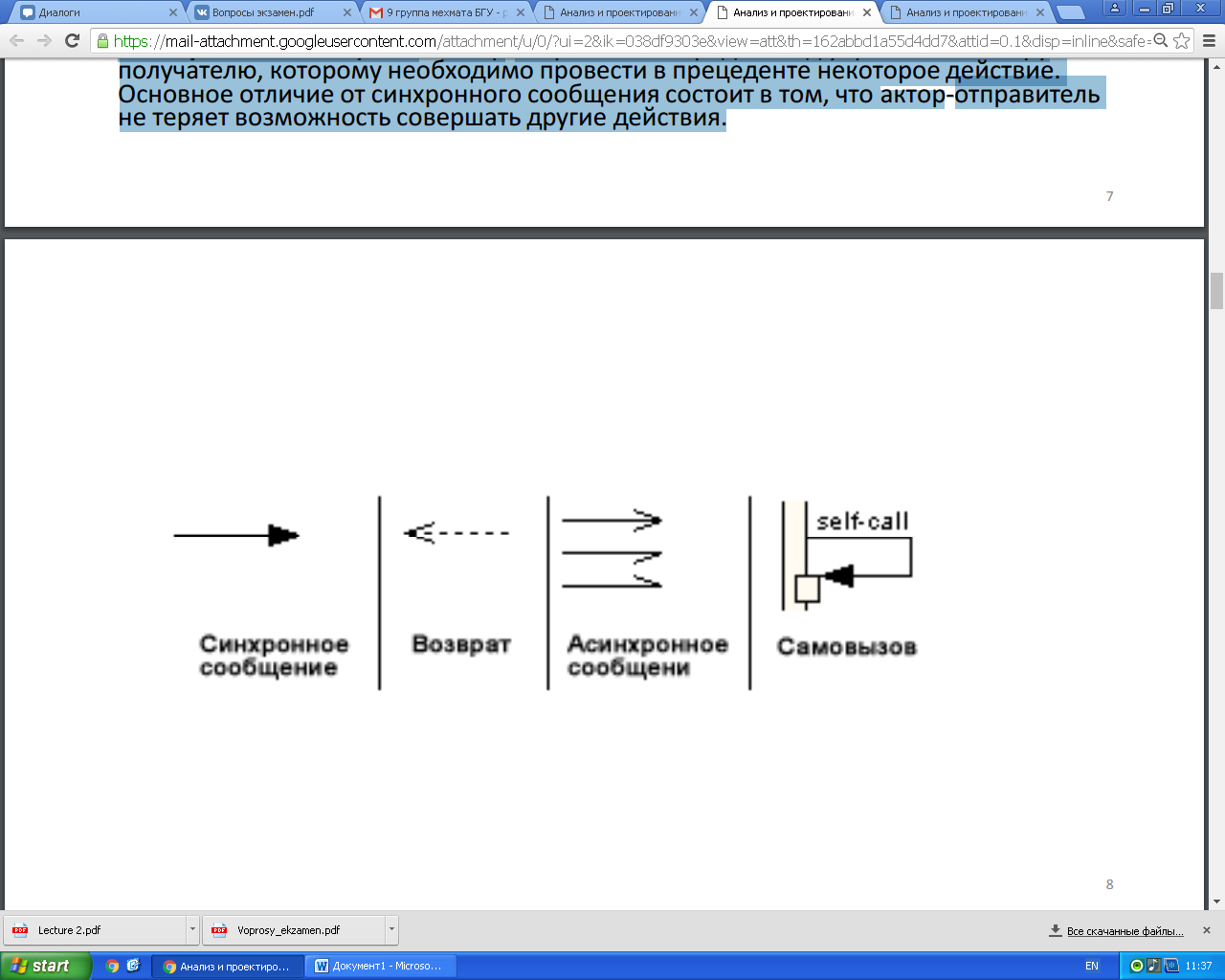
действовать).

*Асинхронное сообщение* - актёр-отправитель передаёт ход управления актёру-

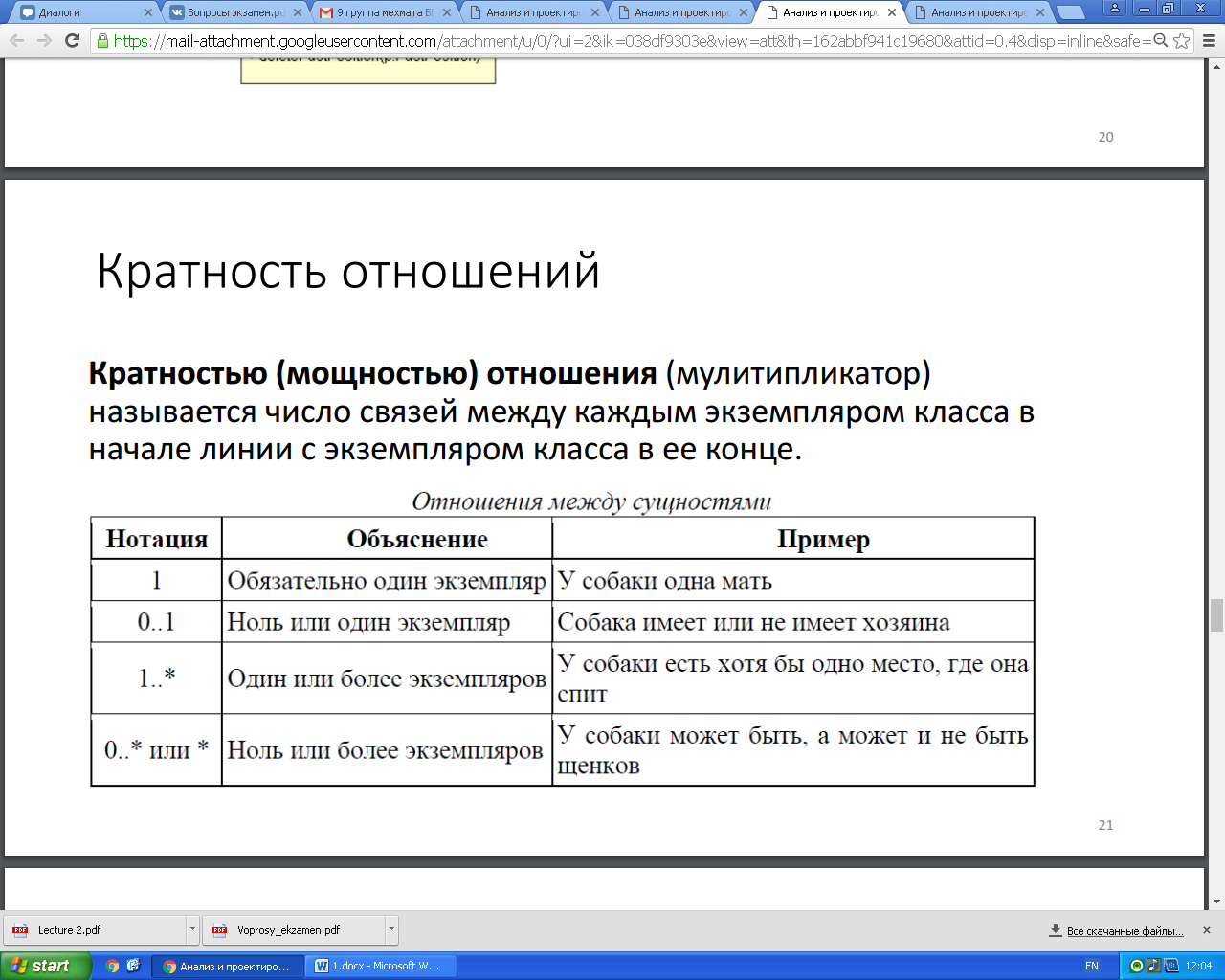
получателю, которому необходимо провести в прецеденте некоторое действие.

Основное отличие от синхронного сообщения состоит в том, что актёр-отправитель

не теряет возможность совершать другие действия.



24. Отношения UML (виды связей)



1. Ассоциации (показывает отношения между объектами-экземплярами класса)

1. Бианарная (связь один к одному)

2. N-нарная (связь один ко многим)

3. Агрегация (агрегация включает в себя *композицию*, в которой при удалении одного объекта уничтожаются и объекты другого класса

2. Обобщения (то же, что и наследование)

3. Реализации

4. Зависимости

**25. Шаблон MVC**

**Model View Controller** - схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

**Модель**: предоставляет данные и реагирует на команды контроллера, изменяя своё состояние

**Представление**: отвечает за отображение данных модели пользователю, реагируя на изменения модели

**Контроллер:** интерпретирует действия пользователя, оповещая модель о необходимости изменений

Назначение: Основная цель применения этой концепции состоит в отделении модели от её визуализации (представления). За счёт такого разделения повышается возможность повторного использования кода. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения.

26. Что такое архитектура ПО? (понимание)

Архитектура ПО – это множество важных решений об организации программной системы, выбор структурных элементов из которых состоит система и их интерфейсов, вместе с поведением определенным как взаимодействие между ними, объединение этих элементов в подсистемы; архитектурный стиль, который помогает их организации – это элементы, их интерфейсы, взаимодействие и состав (Soni, Nord, and Hofmeister, 1995).

27. Язык UML (назначение)

UML (Unified Modeling Language — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

28. Класс, объект, атрибут, метод

**Класс** - описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой.

Классы используются для составления словаря разрабатываемой системы. Это могут быть *абстракции*, являющиеся частью предметной области, либо классы, на которые опирается реализация. С их помощью описываются программные, аппаратные или чисто концептуальные сущности.

С помощью **атрибутов** класса отображается состав и структура данных, которые хранят объекты класса. Каждый атрибут представляется в виде имени и типа, определяющего данные, которые он содержит.

**Метод** - это функция или процедура, принадлежащая какому-то классу или объекту.

**Объект** *-* именованная сущность, которая обладает свойствами и определенным образом проявляет свое поведение. Объекты являются экземплярами класса.

29. Интерфейс, абстракция, наследование, инкапсуляция, полиморфизм

*Абстракция* - это придание объекту характеристик, которые чётко определяют его концептуальные границы, отличая от всех других объектов. Абстракция позволяет представить сложную концепцию в более простой форме *Интерфейс* (от лат. Inter – «между», и face – «поверхность») – семантическая и синтаксическая конструкция в коде программы, используемая для специфицирования услуг, предоставляемых классом или компонентом. Интерфейс определяет границу взаимодействия между классами или компонентами, специфицируя определенную абстракцию, которую осуществляет реализующая сторона. Другими словами, интерфейс – это набор открытых методов, которые определяют поведение класса. Класс, реализующий интерфейс берет на себя обязанность определить методы интерфейса. Интерфейс класса должен составлять согласованную абстракцию. Иначе говоря, открытые методы класса должны быть логически согласованны между собой.

*Наследованием (inheritance)* является процесс, с помощью которого один объект может наследовать свойства другого объекта. Другими словами, объект может приобретать свойства, присущие другому объекту и добавлять к ним определенные свойства и методы, характерные только для него. *Инкапсуляцией* (encapsulation) называется принцип, направленный на объединение данных и кода, с помощью которого осуществляется манипуляция этими данными, а также защита данных от неправильного использования и прямого внешнего доступа. Другими словами, манипуляции данными класса осуществляется с помощью методов этого же класса.

Полиморфизмом (polymorphism) называется механизм, который направлен на использование одного и того же имени метода для осуществления нескольких похожих, но отличающихся задач. Цель полиморфизма в объектно-ориентированном программировании заключается в использовании одного имени для задания общих действий, присущих классу. Другими словами концепцию полиморфизма можно выразить в виде идеи «один интерфейс, множество методов».

30. Диаграмма компонентов и диаграмма развертывания

Диаграмма компонентов, Component diagram — статическая структурная диаграмма, показывает разбиение программной системы на структурные компоненты и связи (зависимости) между компонентами. В качестве физических компонентов могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т. п

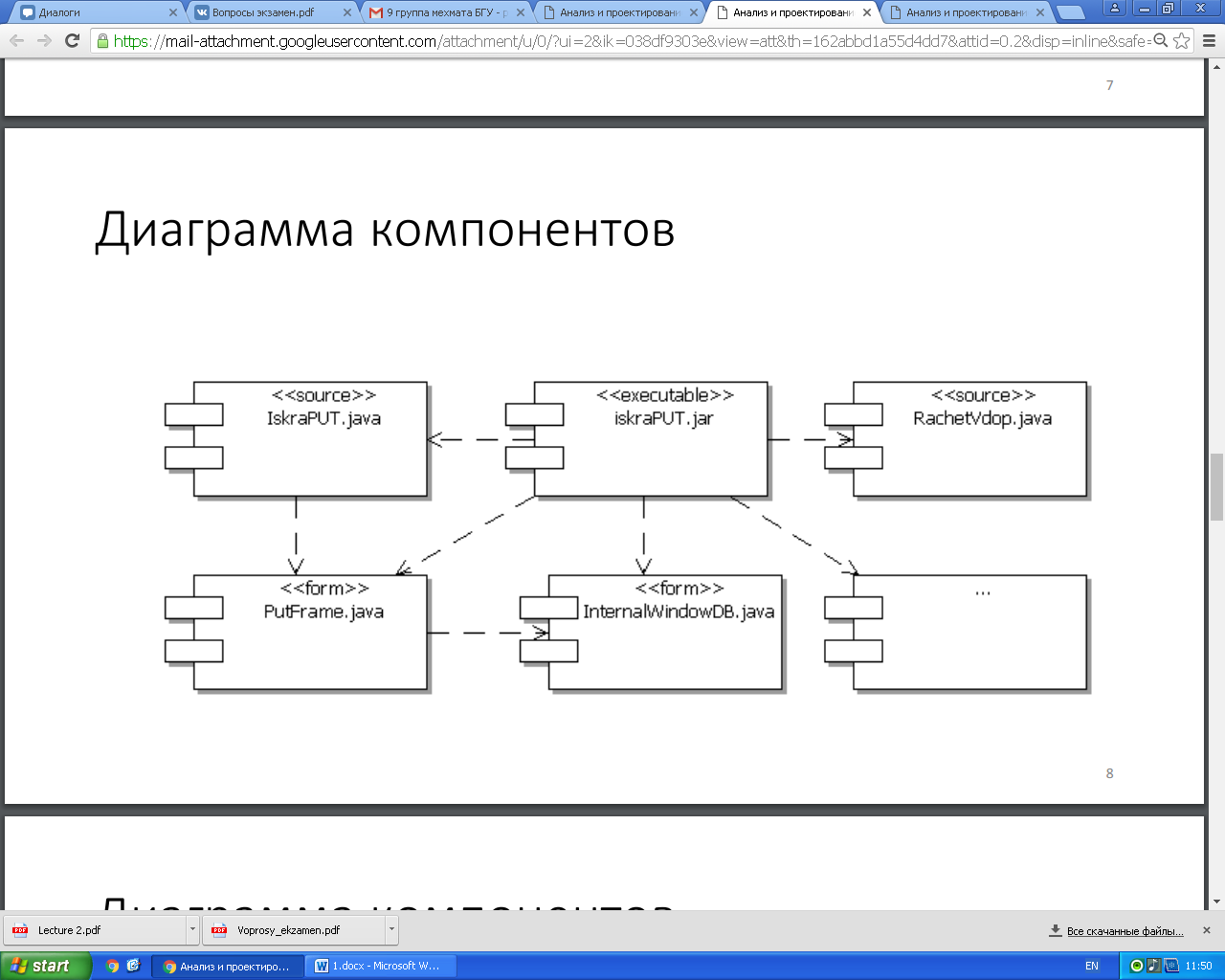




Диаграмма развертывания

Диаграмма развёртывания моделирует физическое развертывание артефактов на узлах.

*Два типа узлов*

*Узлы устройств* — это физические вычислительные ресурсы со своей памятью и сервисами для выполнения программного обеспечения, такие как обычные ПК

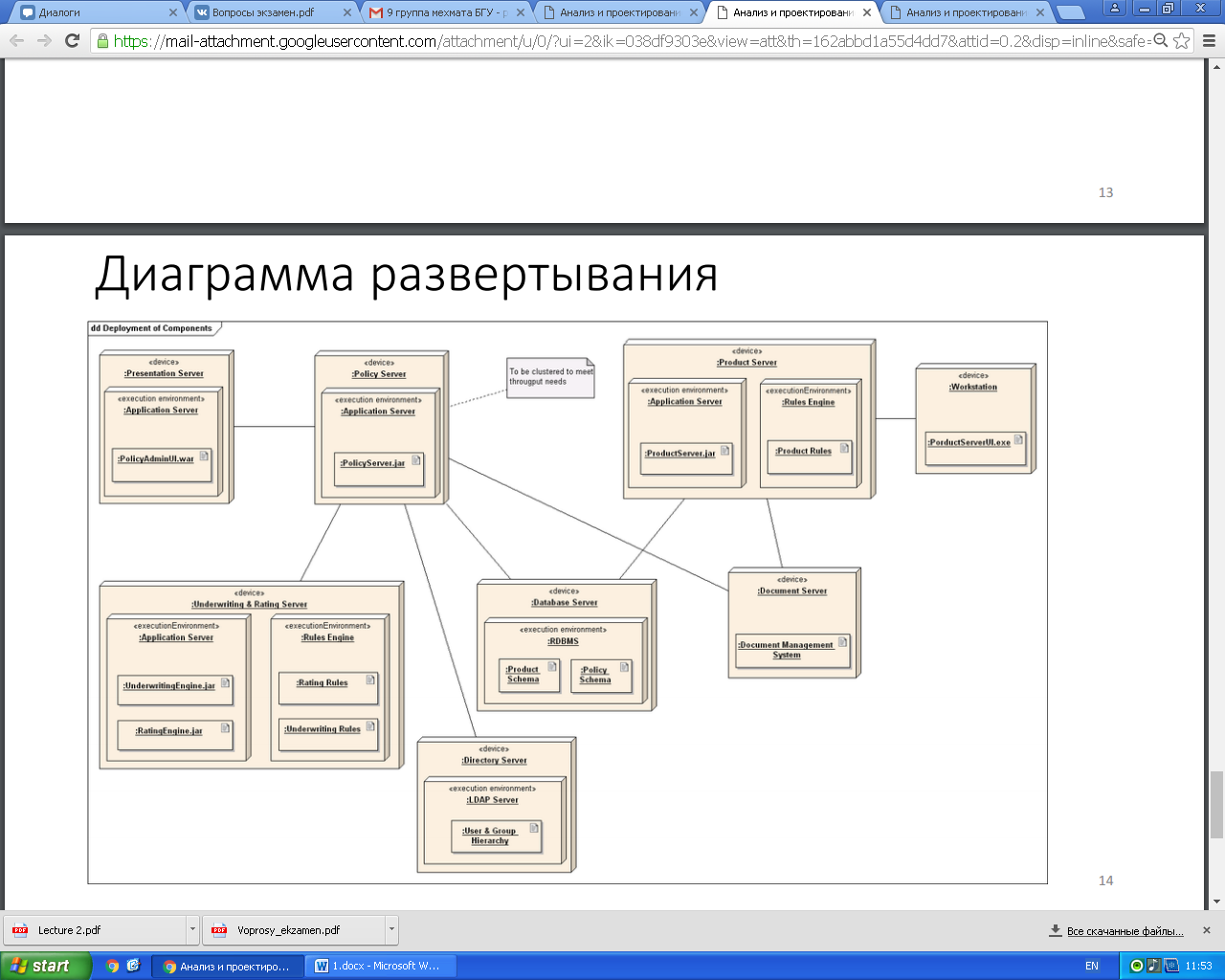
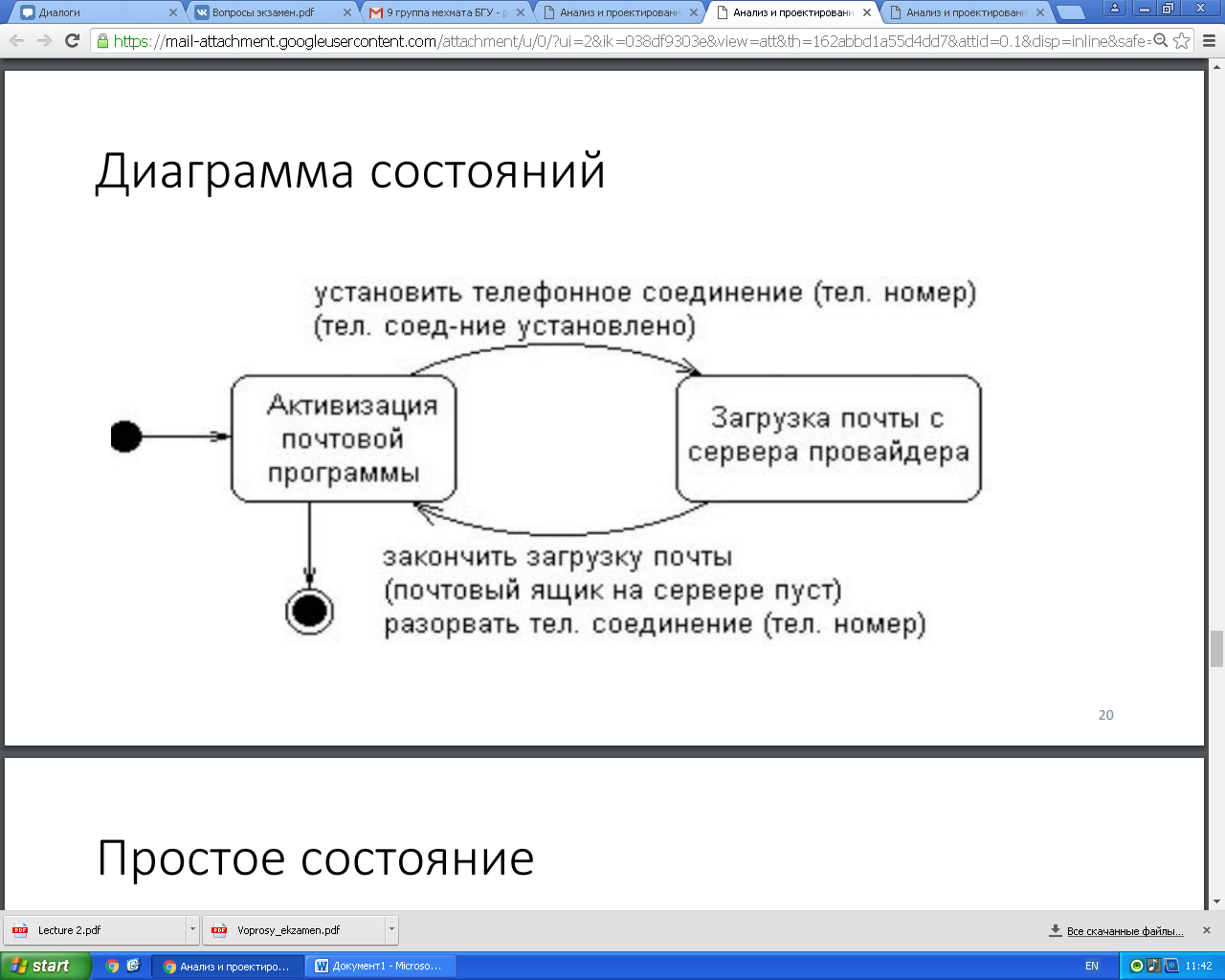
*Узел среды выполнения* — это программный вычислительный ресурс, который работает внутри внешнего узла и который предоставляет собой сервис, выполняющий другие исполняемые программные элементы.

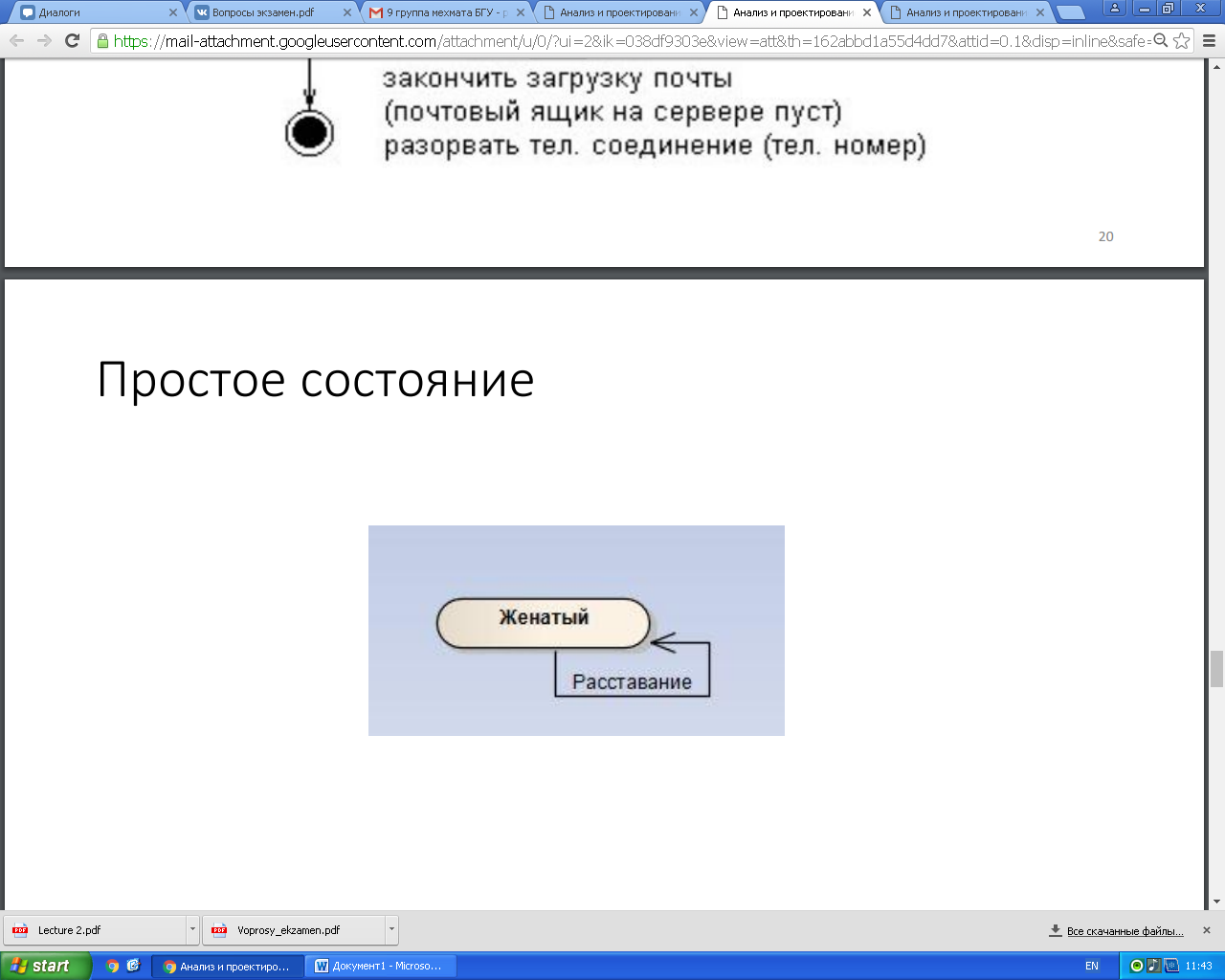
Диаграмма состояний

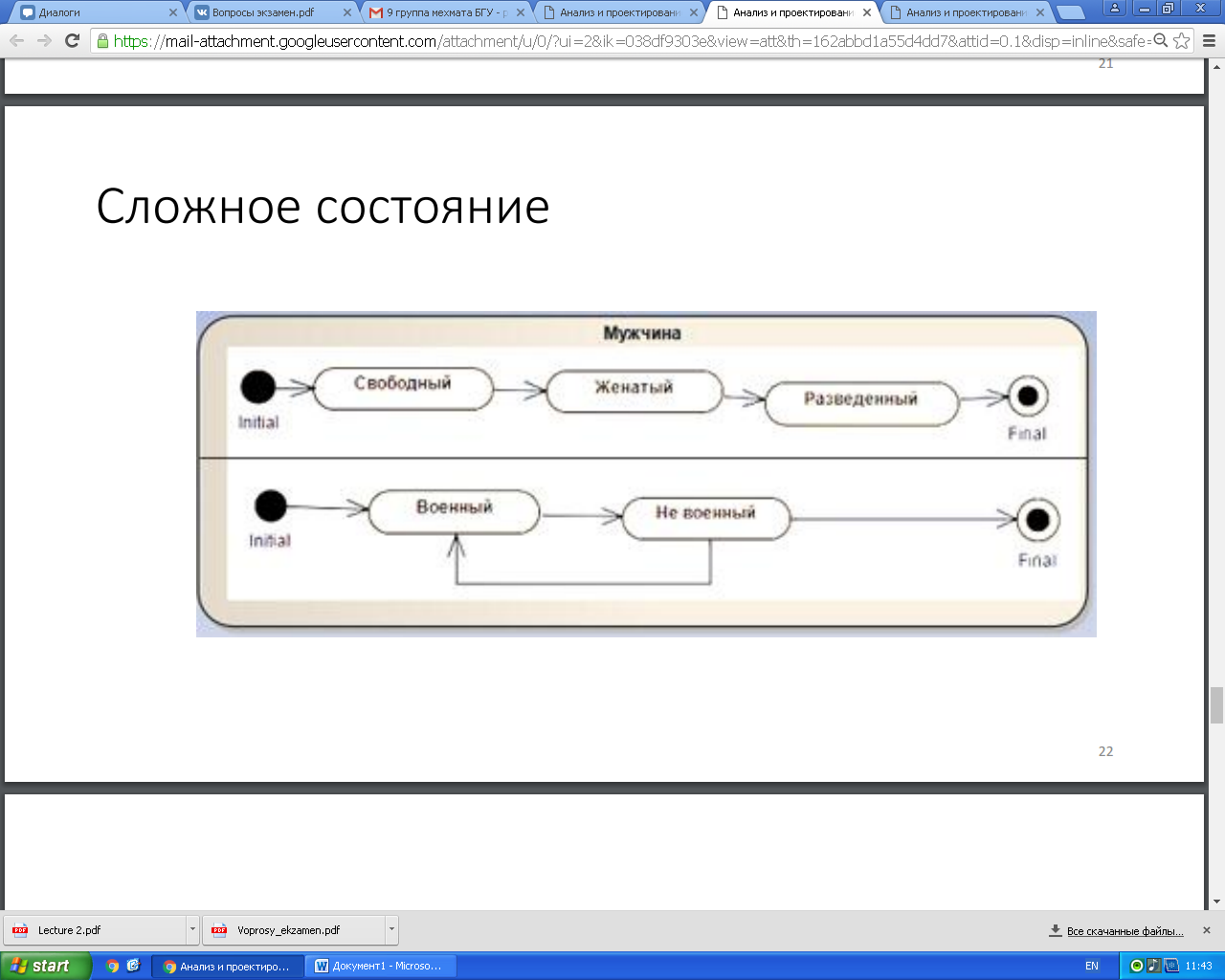
Диаграмма перехода состояний позволяет получить точное, полное и ясное представление о механизме с конечным числом состояний.

Диаграмма состояний – ориентированный граф для конечного автомата, в котором вершины обозначают состояние, а дуги показывают переход между двумя состояниями.

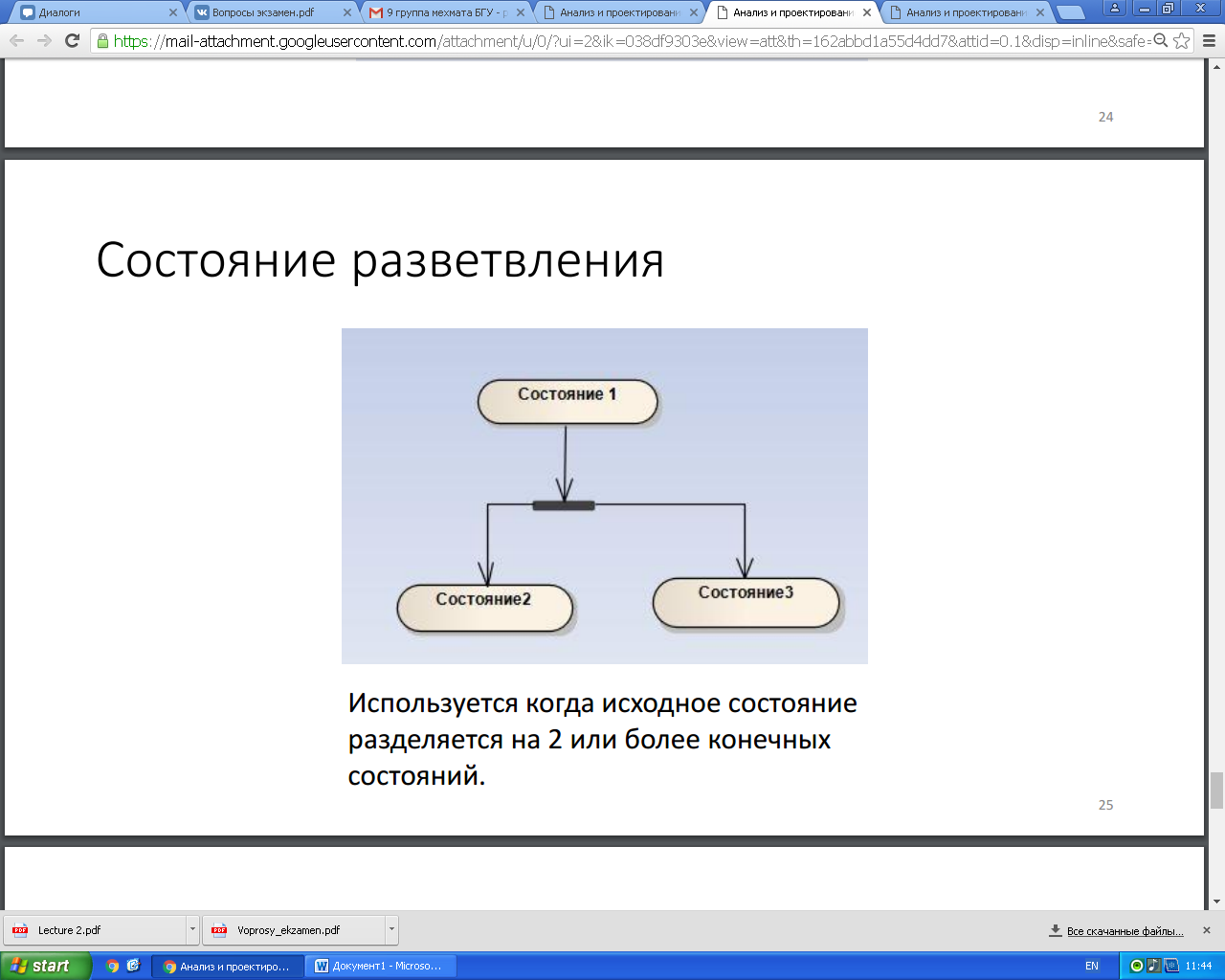
Конечный автомат - это график состояний и переходов между ними, который описывает реакцию объекта.

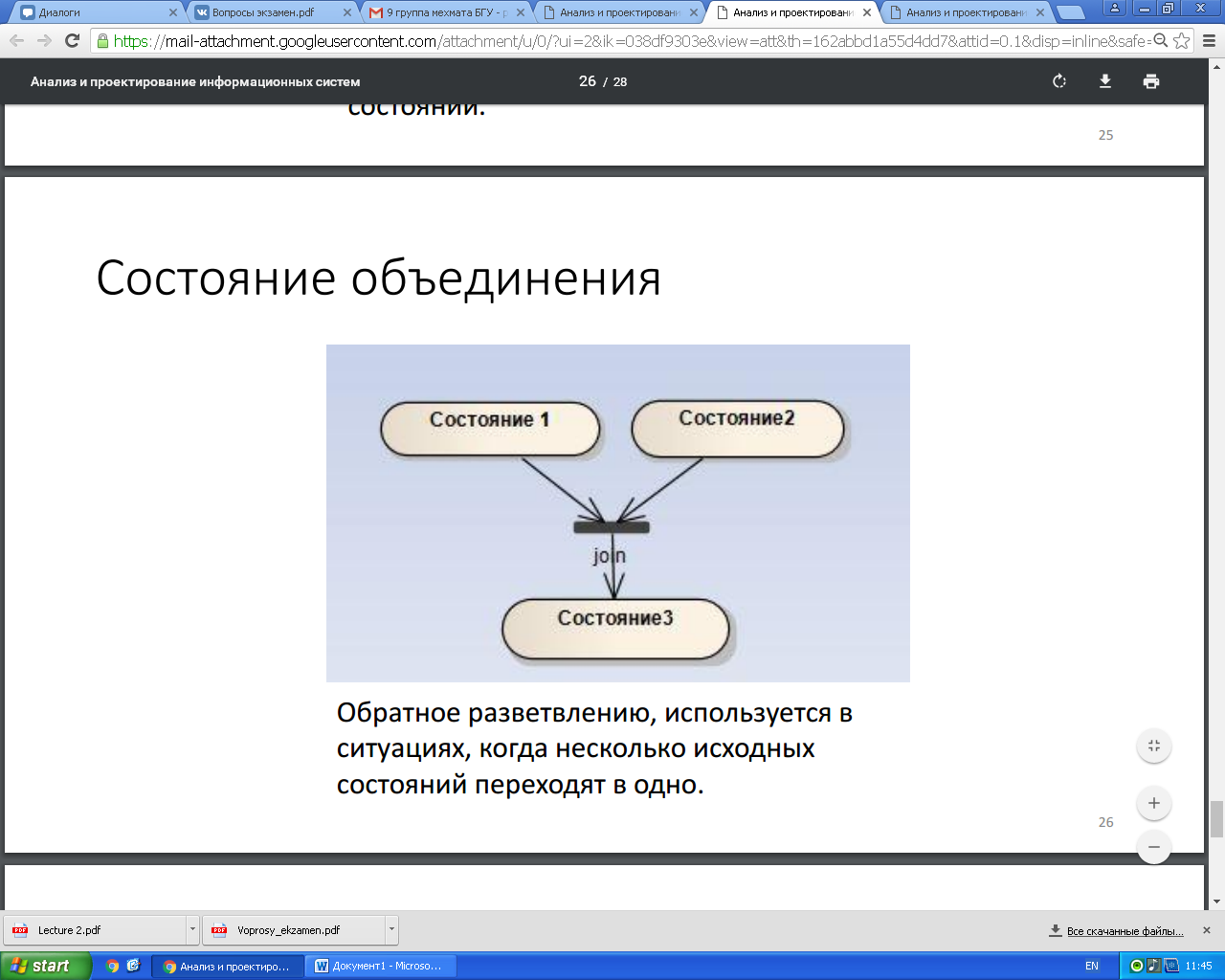












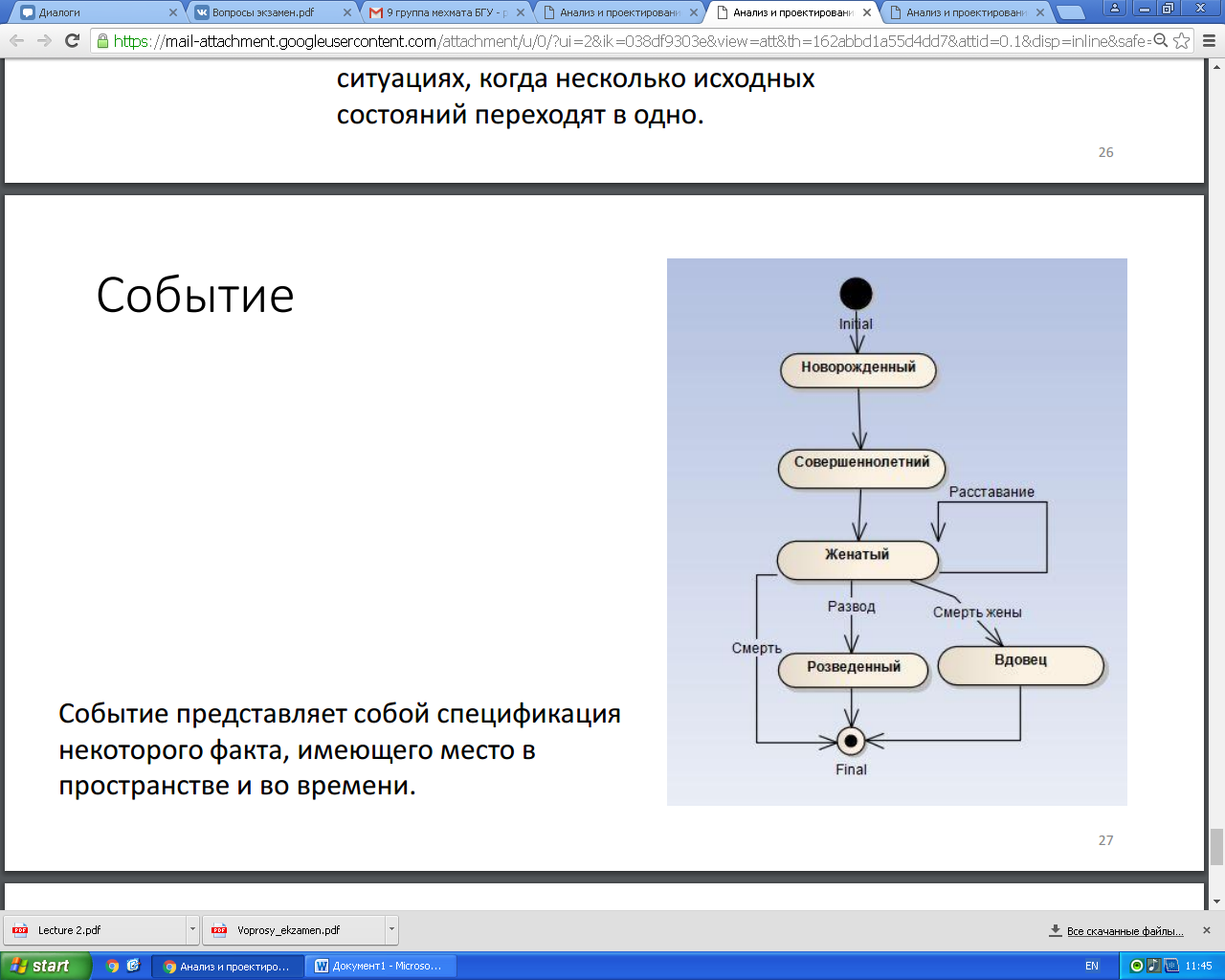


Диаграмма пакетов

Диаграммы пакетов отображают зависимости между пакетами, составляющими модель.

